PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 9/38	A1	(11) International Publication Number: WO 00/29942 (43) International Publication Date: 25 May 2000 (25.05.00)

(21) International Application Number: PCT/SE99/02064

(22) International Filing Date: 12 November 1999 (12.11.99)

(30) Priority Data:
9803901-9 16 November 1998 (16.11.98) SE

(71) Applicant: TELEFONAKTIEBOLAGET LM ERICSSON [SE/SE]; S-126 25 Stockholm (SE).

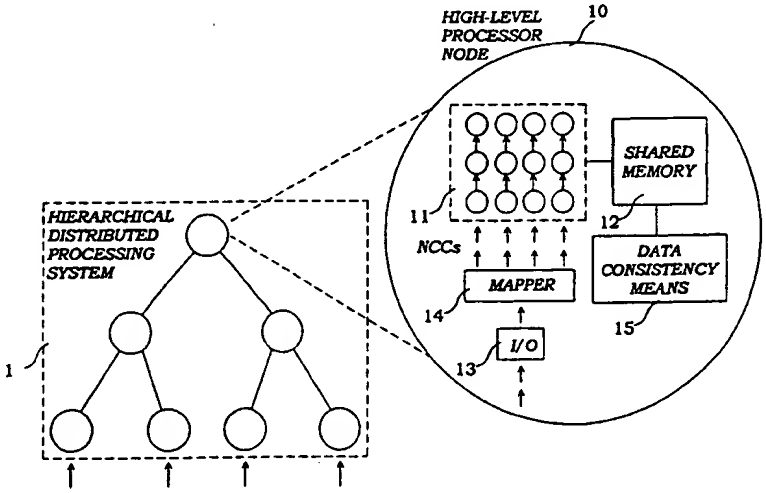
(72) Inventors: HOLMBERG, Per, Anders; Flintbacken 18, S-118 42 Stockholm (SE). KLING, Lars-Örjan; Kummelvägen 17, S-151 52 Södertälje (SE). JOHNSON, Sten, Edward; Lysviksgatan 3, S-123 42 Farsta (SE). SOHONI, Milind; C-147, CSRE Quarters, Hillside, Indian Institute of Technology, Powai, Mombai 400076 (IN). TIKEKAR, Nikhil; C 321, Century Park, Richmond Road, Bangalore 560025 (IN).

(74) Agents: HEDMAN, Anders et al.; Aros Patent AB, P.O. Box 1544, S-751 45 Uppsala (SE).

(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.

(54) Title: CONCURRENT PROCESSING FOR EVENT-BASED SYSTEMS


HIGH-LEVEL PROCESSOR NODE
10
HIERARCHICAL DISTRIBUTED PROCESSING SYSTEM
1
SHARED MEMORY
11
12
NCCs
DATA CONSISTENCY MEANS
MAPPER
14
15
I/O
13

(57) Abstract

According to the invention multiple shared-memory processors (11) are introduced at the highest level or levels of a hierarchical distributed processing system (1), and the utilization of the processors is optimized based on concurrent event flows identified in the system. According to a first aspect, so-called non-commuting categories (NCCs) of events are mapped onto the multiple processors (11) for concurrent execution. According to a second aspect of the invention, the processors (11) are operated as a multiprocessor pipeline, where each event arriving to the pipeline is processed in slices as a chain of internal events which are executed in different stages of the pipeline. A general processing structure is obtained by what is called matrix processing, where non-commuting categories are executed by different sets of processors, and at least one processor set operates as a multiprocessor pipeline in which an external event is processed in slices in different processor stages of the pipeline.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

CONCURRENT PROCESSING FOR EVENT-BASED SYSTEMS

TECHNICAL FIELD OF THE INVENTION

The present invention generally relates to an event-based processing system, and more particularly to a hierarchical distributed processing system as well as a processing method in such a processing system.

BACKGROUND OF THE INVENTION

From a computational point of view, many event-based systems are organized as hierarchical distributed processing systems. For example, in modern telecommunication and data communication networks, each network node normally comprises a hierarchy of processors for processing events from the network. In general, the processors in the hierarchy communicate by message passing, and the processors at the lower levels of the processor hierarchy perform low-level processing of simpler sub-tasks, and the processors at the higher levels of the hierarchy perform high-level processing of more complex tasks.

These hierarchical architectures already exhibit some harnessing of inherent concurrency, but as the number of events to be processed per time unit increases, the higher levels of the processor hierarchy become bottlenecks for further increase in performance. For example, if the processor hierarchy is implemented as a "tree" structure, then the processor at the highest level of the hierarchy becomes the primary bottleneck.

The conventional approach for alleviating this problem mainly relies on the use of higher processor clock frequencies, faster memories and instruction pipelining.

RELATED ART

U.S. Patent 5,239,539 issued to Uchida et al. discloses a controller for controlling the switching network of an ATM exchange by uniformly distributing loads among a plurality of call processors. A main processor assigns originated call processings to the call processors in the sequence of call originations or by the channel identifiers attached to the respective cells of the calls. A switching state controller collects usage information about a plurality of buffers in the switching network, and the call processors perform call processings based on the content of the switching state controller.

The Japanese Patent abstract JP 6276198 discloses a packet switch in which plural processor units are provided, and the switching processing of packets is performed with the units being mutually independent.

The Japanese Patent abstract JP 4100449 A discloses an ATM communication system which distributes signaling cells between an ATM exchange and a signaling processor array (SPA) by STM-multiplexing ATM channels. Scattering of processing loads is realized by switching the signaling cells by means of an STM on the basis of SPA numbers added to each virtual channel by a routing tag adder.

The Japanese Patent abstract JP 5274279 discloses a parallel processing device which is in the form of a hierarchical set of processors, where processor element groups are in charge of parallel and pipeline processing.

SUMMARY OF THE INVENTION

It is an object of the present invention to increase the throughput of event-based hierarchical distributed processing systems. In particular, it is desirable to decongest bottlenecks constituted by high-level processor nodes in hierarchical systems.

It is another object of the invention to provide a processing system, preferably but not necessarily operating as a high-level processor node, which is capable of efficiently processing events based on an event flow concurrency identified in the system.

Yet another object of the invention is to provide a processing system which is capable of exploiting concurrency in the event flow while still allowing reuse of existing application software.

Still another object of the invention is to provide a method for efficiently processing events in a hierarchical distributed processing system.

These and other objects are met by the invention as defined by the accompanying patent claims.

A general idea according to the invention is to introduce multiple shared-memory processors at the highest level or levels of a hierarchical distributed processing system, and optimize the utilization of the multiple processors based on concurrent event flows identified in the system.

According to a first aspect of the invention, the external event flow is divided into concurrent categories, referred to as non-commuting categories, of events and these non-commuting categories are then mapped onto the multiple processors for concurrent execution. Non-commuting categories are generally groupings of events where the order of events must be preserved within a category, but where there are no ordering requirements between categories. For example, a non-commuting category may be defined by events generated by a predetermined source such as a particular input port, regional processor or hardware device connected to the system. Each non-commuting category of events is assigned to a predetermined set of one or more processors, and internal events generated by a predetermined processor set are fed back to the same processor set in order to preserve the non-commuting category or categories assigned to that processor set.

According to a second aspect of the invention, the multiple processors are operated as a multiprocessor pipeline having a number of processor stages, where each external event arriving to the pipeline is processed in slices as a chain of internal events which are executed in different stages of the pipeline. In general, each pipeline stage is executed in one of the processors, but a given processor may execute more than one stage of the pipeline. A particularly advantageous way of realizing a multiprocessor pipeline is to allocate a cluster of software blocks/classes in the shared memory software to each processor, where each event is targeted for a particular block, and then distribute the events onto the processors based on this allocation.

A general processing structure is obtained by what is called matrix processing, where non-commuting categories are executed by different sets of processors, and at least one processor set is in the form of an array of processors which operates as a multiprocessor pipeline in which an external event is processed in slices in different processor stages of the pipeline.

In a shared memory system, the entire application program and data are accessible to all the shared-memory processors in the system. Accordingly, data consistency must be assured when global data are manipulated by the processors.

According to the invention, data consistency can be assured by locking global data to be used by a software task that is executed in response to an event, or in the case of an object-oriented software design locking entire software blocks/objects. If processing of an event requires resources from more than one block, then the locking approach may give rise to deadlocks, where tasks are mutually locking each other. Therefore, deadlocks are detected and roll-back performed to ensure progress, or alternatively deadlocks are completely avoided by seizing all blocks required by a task before initiating execution of the task.

Another approach for assuring data consistency is based on parallel execution of tasks, where access collisions between tasks are detected and an executed task, for which a collision is detected, is rolled-back and restarted. Collisions are either detected based on variable usage markings, or alternatively detected based on address comparison where read and write addresses are compared.

By marking larger areas instead of individual data, a more coarse-grained collision check is realized.

The solution according to the invention substantially increases the throughput capacity of the processing system, and for hierarchical processing systems the high-level bottlenecks are efficiently decongested.

By using shared-memory multiprocessors and providing appropriate means for assuring data consistency, application software already existing for single-processor systems may be reused. In many cases, millions of lines of code are already available for single-processor systems such as single-processor nodes at the highest level of hierarchical processing systems. In the case of implementing the multiple processors using standard off-the-shelf microprocessors, all of the existing application software can be reused by automatically transforming the application software and possibly modifying the virtual machine/operating system of the system to support multiple processors. On the other hand, if the multiple processors are implemented as specialized hardware of proprietary design, the application software can be directly migrated to the multiprocessor environment. Either way, this saves valuable time and reduces the programming costs compared to designing the application software from scratch.

The invention offers the following advantages:

- Increased throughput capacity;
- Decongestion of bottlenecks;
- Allows reuse of already existing application software; especially in the case of object-oriented designs.

Other advantages offered by the present invention will be appreciated upon reading of the below description of the embodiments of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention, together with further objects and advantages thereof, will be best understood by reference to the following description taken together with the accompanying drawings, in which:

Fig. 1 is a schematic diagram of a hierarchical distributed processing system with a high-level processor node according to the invention;

Fig. 2 is a schematic diagram of a processing system according to a first aspect of the invention;

Fig. 3 illustrates a particular realization of a processing system according to the first aspect of the invention;

Fig. 4 is a schematic diagram of a simplified shared-memory multiprocessor with an object-oriented design of the shared-memory software;

Fig. 5A is a schematic diagram of a particularly advantageous processing system according to a second aspect of the invention;

Fig. 5B illustrates a multiprocessor pipeline according to the second aspect of the invention;

Fig. 6 illustrates the use of locking of blocks/objects to assure data consistency;

Fig. 7 illustrates the use of variable marking to detect access collisions;

Fig. 8A illustrates a prior art single-processor system from a stratified viewpoint;

Fig. 8B illustrates a multiprocessor system from a stratified viewpoint; and

Fig. 9 is a schematic diagram of a communication system in which at least one processing system according to the invention is implemented.

DETAILED DESCRIPTION OF EMBODIMENTS OF THE INVENTION

Throughout the drawings, the same reference characters will be used for corresponding or similar elements.

Fig. 1 is a schematic diagram of a hierarchical distributed processing system with a high-level processor node according to the invention. The hierarchical distributed processing system 1 has a conventional tree structure with a number of processor nodes distributed over a number of levels of the system hierarchy. For example, hierarchical processing systems can be found in telecommunication nodes and routers. Naturally the high-level processor nodes, and especially the processor node at the top, become bottlenecks as the number of events to be processed by the processing system increases.

An efficient way of decongesting such bottlenecks according to the invention includes using multiple shared-memory processors 11 at the highest level or levels of the hierarchy. In Fig. 1, the multiple processors are illustrated as implemented at the top node 10. Preferably, the multiple shared-memory processors 11 are realized in the form of a standard microprocessor based multiprocessor system. All processors 11 share a common memory, the so-called shared memory 12. In general, external, asynchronous events bound for the high-level processor node 10 first arrive to an input/output (I/O) unit 13, from which they are forwarded to a mapper or distributor 14. The mapper 14 maps/distributes the events to the processors 11 for processing.

Based on an event flow concurrency identified in the hierarchical processing system 1, the external flow of events to the processor node 10 is divided into a number of concurrent categories, hereinafter referred to as non-commuting categories (NCCs), of events. The mapper 14 makes sure that each NCC is assigned to a predetermined set of one or more of the processors 11, thus enabling concurrent processing and optimized utilization of the multiple processors. The mapper 14 could be implemented in one or more of the processors 11, which then preferably are dedicated to the mapper.

The non-commuting categories are groupings of events where the order of events must be preserved within a category, but where there are no ordering requirements on processing events from different categories. A general requirement for systems where the information flow is governed by protocols is that certain related events must be processed in the received order. This is the invariant of the system, no matter how the system is implemented. The identification of proper NCCs and the concurrent processing of the NCCs guarantee that the ordering requirements imposed by the given system protocols are met, while at the same time the inherent concurrency in the event flow is exploited.

If an external event can be processed or executed in "slices" as a chain of events, alternate or further concurrent execution is possible by operating one or more of the processor sets as multiprocessor pipelines. Each external event arriving to a multiprocessor pipeline is thus processed in slices, which are executed in different processor stages of the multiprocessor pipeline.

Consequently, a general processing structure is obtained by what is called matrix processing, where NCCs are executed by different sets of processors, and at least one of the processor sets operates as a multiprocessor pipeline. It should be understood that some of the elements of the logical "matrix" of processors shown in Fig. 1 may be empty. Reducing the logical matrix of processors shown in Fig. 1 to a single row of processors gives pure NCC processing, and reducing the matrix to a single column of processors gives pure event-level pipeline processing.

The computation for an event-based system is generally modeled as a state machine, where an input event from the external world changes the state of the system and may result in output events. If each non-commuting category/pipeline stage could be processed by an independent/disjoint state machine, there would not be any sharing of data between the various state machines. But given that there are global resources, which are represented by global states or variables, the operation on a given global state normally has to be "atomic" with only one processor, which executes part of the system state machine, accessing a given global state at a time. The need for so-called sequence-dependency checks is eliminated because of the NCC/pipeline-based execution.

For a better understanding, consider the following example. Assume that a certain set of global variables is responsible for allocation of resources such as free channels towards another node for communication. Then, for two asynchronous jobs of different NCCs, the order in which they request a free channel doesn't matter - the first to ask will get the first channel meeting the selection criterion, while the second one gets the next available channel meeting the criterion. What is important is that while the channel selection is in progress for one of the jobs, the other job should not interfere. The access to the global variable or variables responsible for the channel allocation must be "atomic" (although it is possible, in special cases, to even parallelise the channel search).

Another example involves two jobs, of different NCCs, that need to increment a counter. It doesn't matter which job that increments the counter first, but as long as one of the jobs is operating on the countei variable to increment it (read its current value and add one to it) the other job should not interfere.

In a shared-memory system, the entire application program space and data space in the shared memory 12 are accessible to all the processors. Consequently, it is necessary to assure data consistency as the processors need to manipulate global variables common to all of or at least more than one of the processors. This is accomplished by the data consistency means schematically indicated by reference numeral 15 in Fig. 1.

In the following, NCC processing as a first aspect of the invention, event-level pipeline processing as a second aspect of the invention as well as procedures and means for assuring data consistency will be described.

NCC processing

Fig. 2 is a schematic diagram of an event-driven processing system according to a first aspect of the invention. The processing system comprises a number of shared-memory processors P1 to P4, a shared memory 12, an I/O-unit 13, a distributor 14, data consistency means 15 and a number of independent parallel event queues 16.

The I/O-unit 13 receives incoming external events and outputs outgoing events. The distributor 14 divides the incoming events into non-commuting categories (NCCs) and distributes each NCC to a predetermined one of the independent event queues 16. Each one of the event queues is connected to a respective one of the processors, and each processor sequentially fetches or receives events from its associated event queue for processing. If the events have different priority levels, this has to be considered so that the processors will process events in order of priority.

By way of example, consider a hierarchical processing system with a central high-level processor node and a number of lower-level processors, so-called regional processors, where each regional processor in turn serves a number of hardware devices. In such a system, the events originating from the hardware devices and the events coming from the regional processors that serve a group of devices meet the conditions imposed by the ordering requirements that are defined by the given protocols (barring error conditions which are protected by processing at a higher level). So, events from a particular device/regional processor form a non-commuting category. In order to preserve a non-commuting category, each device/regional processor must always feed its events to the same processor.

In telecommunication applications for example, a sequence of digits received from a user, or a sequence of ISDN user part messages received for a trunk device must be processed in the received order, whereas sequences of messages received for two independent trunk devices can be processed in any order as long as the sequencing for individual trunk devices is preserved.

In Fig. 2 it can be seen that events from a predetermined source S1, for example a particular hardware device or input port, are mapped onto a predetermined processor P1, and events from another predetermined source S2, for example a particular regional processor, are mapped onto another predetermined processor P3. Since, the number of sources normally exceeds the number of shared-memory processors by far, each processor is usually assigned a number of sources. In a typical telecom/datacom application, there could be 1024 regional processors communicating with a single central processor node. Mapping regional processors onto the multiple shared-memory processors in the central node in a load balanced way means that each shared-memory processor roughly gets 256 regional processors (assuming that there are 4 processors in the central node, and all regional processors generate the same load). In practice however, it might be beneficial to have an even finer granularity, mapping hardware devices such as signaling devices, subscriber terminations, etc. to the central node processors. This generally makes it easier to obtain load balance. Each regional processor in a telecom network might control hundreds of hardware devices. So instead of mapping 10,000 or more hardware devices onto a single processor, which of course handles the load in a time-shared manner, the solution according to the invention is to map the hardware devices onto a number of shared-memory processors in the central node, thus decongesting the bottleneck in the central node.

A system such as the AXE Digital Switching System of Telefonaktiebolaget LM Ericsson that processes an external event in slices connected by processor-to-processor (CP-to-CP) signals or so-called internal events, might impose its own sequencing requirement in addition to the one imposed by protocols. Such CP-to-CP signals for an NCC must be processed in the order in which they are generated (unless superseded by a higher priority signal generated by the last slice under execution). This additional sequencing requirement is met if each CP-to-CP signal (internal event) is processed in the same processor in which it is generated, as indicated in Fig. 2 by the dashed lines from the processors to the event queues. So, internal events are kept within the same NCC by feeding them back to the same processor or processor set that generated them - hence guaranteeing that they are processed in the same order in which they were generated.

Normally, the representation of the events as seen by the processing system are signal messages. In general, each signal message has a header and a signal body. The signal body includes information necessary for execution of a software task. For example, the signal body includes, implicitly or explicitly, a pointer to software code/data in the shared memory as well as the required input operands. In this sense, the event signals are self-contained, completely defining the corresponding task. Consequently, the processors P1 to P4 independently fetch and process events to execute corresponding software tasks, or jobs, in parallel. A software task is also referred to as a job, and throughout the disclosure, the terms task and job are used interchangeably. During parallel task execution, the processors need to manipulate global data in the shared memory. In order to avoid data inconsistencies, where several processors access and manipulate the same global data (during the lifetime of a job), the data consistency means 15 must make sure that data consistency is assured at all times. The invention makes use of two basic procedures for assuring data consistency when global data is manipulated by the processors during parallel task execution:

- Locking: Each processor normally comprises means, forming part of the data consistency means 15, for locking the global data to be used by a corresponding task before starting execution of the task. In this way, only the processor that has locked the global data can access it. Preferably, the locked data is released at the end of execution of the task. This approach means that if global data is locked by a processor, and another processor wants to access the same data that other processor has to wait until the locked data is released. Locking generally implies waiting times (wait/stall on a locked global state) which limits the amount of parallel processing to some degree (concurrent operations on different global states at the same time is of course allowed).

- Collision detection and roll-back: Software tasks are executed in parallel, and access collisions are detected so that one or more executed tasks for which collisions are detected can be rolled-back and restarted. Collision detection is generally accomplished by a marker method or an address comparison method. In the marker method, each processor comprises means for marking the use of variables in the shared memory, and variable access collisions are then detected based on the markings. Collision detection generally has a penalty due to roll-backs (resulting in wasted processing).

Which approach to choose depends on the application, and has to be selected on a case-to-case basis. A simple rule of thumb is that locking based data consistency might be more suitable for database systems, and collision detection more beneficial for telecom and datacom systems. In some applications, it may even be advantageous to use a combination of locking and collision detection.

Locking and collision detection as means for assuring data consistency will be described in more detail later on.

Fig. 3 illustrates a particular realization of a processing system according to the first aspect of the invention. In this realization, the processors P1 to P4 are symmetrical multiprocessors (SMPs) where each processor has its own local cache C1 to C4, and the event queues are allocated in the shared memory 12 as dedicated memory lists, preferably linked lists, EQ1 to EQ4.

As mentioned before, each event signal generally has a header and a signal body. In this case, the header includes an NCC tag (implicit or explicit) which is representative of the NCC to which the corresponding event belongs. The distributor 14 distributes an incoming event to one of the event queues EQ1 to EQ4 based on the NCC tag included in the event signal. By way of example, the NCC tag may be a representation of the source, such as an input port, regional processor or hardware device, from which the event originates. Assume that an event received by the I/O-unit 13 comes from a particular hardware device and that this is indicated in the tag included in the event signal. The distributor 14 then evaluates the tag of the event, and distributes the event to a predetermined one of the shared-memory allocated event queues EQ1 to EQ4 based on a pre-stored event-dispatch table or equivalent. Each one of the processors P1 to P4 fetches events from its own dedicated event queue in the shared memory 12 via its local cache to process and terminate the events in a sequence. The event-dispatch table could be modified from time to time to adjust for long-term imbalances in traffic sources.

Of course, the invention is not limited to symmetrical multiprocessors with local caches. Other examples of shared-memory systems include shared-memory without cache, shared memory with common cache as well as shared memory with mixed cache.

Example of object-oriented design

Fig. 4 is a schematic diagram of a simplified shared-memory multiprocessor system having an object-oriented design of the shared-memory software. The software in the shared memory 12 has an object-oriented design, and is organized as a set of blocks B1 to Bn or classes. Each block/object is responsible for executing a certain function or functions. Typically, each block/object is split into two main sectors - a program sector where the code is stored and a data sector where the data is stored. The code in the program sector of a block can only access and operate on data belonging to the same block. The data sector in turn is preferably divided into two sectors as well - a first sector of "global" data comprising a number of global variables GV1 to GVn, and a second sector of for example "private" data such as records R1 to Rn, where each record typically comprises a number of record variables RV1 to RVn as illustrated for record Rx. Each transaction is typically associated with one record in a block, whereas global data within a block could be shared by several transactions.

In general, a signal entry into a block initiates processing of data within the block. On receiving an event, external or internal, each processor executes code in the block indicated by the event signal and operates on global variables and record variables within that block, thus executing a software task. The execution of a software task is indicated in Fig. 4 by a wavy line in each of the processors P1 to P4.

In the example of Fig. 4, the first processor P1 executes code in software block B88. A number of instructions, of which only instructions I20 to I23 are illustrated, are executed, and each instruction operates on one or more variables within the block. For example, instruction I20 operates on record variable RV28 in record R1, instruction I21 operates on record variable RV59 in record R5, instruction I22 operates on the global variable GV43 and instruction I23 operates on the global variable GV67. Correspondingly, the processor P2 executes code and operates on variables in block B1, the processor P3 executes code and operates on variables in block B8 and the processor P4 executes code and operates on variables in block B99.

An example of a block-oriented software is the PLEX (Programming Language for Exchanges) software of Telefonaktiebolaget LM Ericsson, in which the entire software is organized in blocks. Java applications are examples of truly object-oriented designs.

Event-level pipelining

As mentioned earlier, some systems process external events in "slices" connected by internal events (e.g. CP-to-CP buffered signals).

According to a second aspect of the invention concurrent execution is accomplished by operating at least a set of the multiple shared-memory processors as a multiprocessor pipeline where each external event is processed in slices as a chain of events which are executed in different processor stages of the pipeline. The sequencing requirement of processing signals in order of their creation will be guaranteed as long as all the signals generated by a stage are fed to the subsequent stage in the same order as they are generated. Any deviation from this rule will have to guarantee racing-free execution. If execution of a given slice results in more than one signal, then these signals either have to be fed to the subsequent processor stage in the same order as they are generated, or if the signals are distributed to two or more processors it is necessary to make sure that the resulting possibility of racing is harmless for the computation.

Now a particular realization of a multiprocessor pipeline according to the second aspect of the invention will be described with reference to Figs. 5A-B.

Fig. 5A is a schematic diagram of an event-driven processing system according to the second aspect of the invention. The processing system is similar to that shown in Fig. 2. However, internal events generated by a processor that is part of the multiprocessor pipeline 11 are not necessarily fed back to the same processor, but can be fed to any of the processors, as indicated by the dashed lines that originate from the processors P1 to P4 and terminate on the bus to the event queues 16.

In an object-oriented software design, the software in the shared memory is organized into blocks or classes as described above in connection with Fig. 4, and on receiving an external event the corresponding processor executes code in a block/object and may generate results in the form of an internal event towards another block/object. When this internal event comes for execution it is executed in the indicated block/object and might generate another internal event towards some other block/object. The chain usually dies after a few internal events. In telecommunication applications for example, each external event may typically spawn 5-10 internal events.

A realization of a multiprocessor pipeline customized for object-oriented software design is to allocate clusters of software blocks/classes to the processors. In Fig. 2, clusters CL1 to CLn of blocks/classes in the shared memory 12 are schematically indicated by dashed boxes. As can be seen from Fig. 2, one of the clusters CL1 is allocated to the processor P2 as indicated by the solid line interconnecting CL1 with P2, and another cluster CL2 is allocated to the processor P4 as indicated by the dashed line interconnecting CL2 with P4. In this way, each cluster of blocks/classes within the shared memory 12 is allocated to a predetermined one of the processors P1 to P4, and the allocation scheme is implemented in a look-up table 17 in the distributor 14 and in a look-up table 18 in the shared memory 12. Each of the look-up tables 17, 18 links a target block to each event based on e.g. the event ID, and associates each target block to a predetermined cluster of blocks. The distributor 14 distributes external events to the processors according to the information in the look-up table 17. The look-up table 18 in the shared memory 12 is usable by all of the processors P1 to P4 to enable distribution of internal events to the processors. In other words, when a processor generates an internal event, it consults the look-up table 18 to determine i) the corresponding target block based on e.g. the event ID, ii) the cluster to which the identified target block belongs, and iii) the processor to which the identified cluster is allocated, and then feeds the internal event signal to the appropriate event queue. It is important to note that normally each block belongs to one and only one cluster, although an allocation scheme with overlapping clusters could be implemented in a slightly more elaborate way by using information such as execution state in addition to the event ID.

As indicated in Fig. 5B, mapping clusters of blocks/classes to processors automatically causes pipelined execution – let us say the external event EE is directed to block A, which is allocated to processor P1, then the internal event IE generated by this block is directed to block B, which is allocated to processor P2, then the internal event IE generated by this block is directed to block C, which is allocated to processor P4, and the internal event IE generated by this block is directed to block D which is allocated to processor P1. Hence, we logically have a pipeline with a number of processor stages. Here it is assumed that blocks A and D are part of a cluster mapped to processor P1, whereas block B is part of a cluster mapped to processor P2 and block C is part of a cluster mapped to processor P4. Each stage in the pipeline is executed in one processor, but a given processor may execute more than one stage of the pipeline.

A variation includes mapping events that require input data from a predetermined data area in the shared memory 12 to one and the same predetermined processor set.

It should be understood that when a processor stage in the multiprocessor pipeline has executed an event belonging to a first chain of events, and sent the resulting internal event signal to the next processor stage, it is normally free to start processing an event from the next chain of events, thus improving the throughput capacity.

For maximum gain, the mapping of pipeline stages to the processors should be such that all the processors are equally loaded. Therefore, the clusters of blocks/classes are partitioned according to an "equal load" criterion. The amount of time spent in each cluster can be known for example from a similar application running on a single processor, or could be monitored during run-time to enable re-adjustment of the partitioning. In the case a block generates more than one internal event in response to an input event, and each generated event is directed to different blocks, a "no racing" criterion along with the "equal load" criterion is required to prevent an internal event generated "later" than another event from being executed "earlier".

Of course, it is possible to process an external event without splitting it up into slices, but splitting it up allows structured program development/maintenance and also allows pipelined processing.

Also, the same processing of an external event can be performed in a few big slices or many small slices.

As mentioned above, there are two basic procedures for assuring data consistency when global data is manipulated by the processors during parallel task execution - i) locking and ii) collision detection and roll-back.

Locking as a means of assuring data consistency

When implementing locking for the purpose of assuring data consistency, each processor, in executing a task, generally locks the global data to be used by the task before starting execution of the task. In this way, only the processor that has locked the global data can access it.

Locking is very suitable for object-oriented designs as the data areas are clearly defined, allowing specific data sectors of a block or an entire block to be locked. Lacking a general characterization of global data as it is normally not possible to know which part of the global data in a block that will be modified by a given execution sequence or task, locking the entire global data sector is a safe way of assuring data consistency. Ideally, just protecting the global data in each block is sufficient, but in many applications there are certain so-called "across record" operations that also need to be protected. For example, the operation of selecting a free record will go through many records to actually find a free record. Hence locking the entire block protects everything. Also, in applications where the execution of a buffered signal could span multiple blocks connected by so-called direct/combined signals (direct jumps from one block to another) with possibility of loops (visiting one block more than once before EXIT), it is necessary not to release a locked block until the end of execution of the task.

The use of NCCs will generally minimize "shared states" between the multiple processors and also improve the cache hit rate. In particular, by mapping for example functionally different regional processors/hardware devices such as signaling devices and subscriber terminations in a telecommunication system to different processors in the central node, simultaneous processing of different access mechanisms with little or no wait on locked blocks is allowed since different access mechanisms are normally processed in different blocks till the processing reaches the late stages of execution.

Fig. 6 illustrates the use of locking of blocks/objects to assure data consistency. Consider three different external events EEx, EEy and EEz being directed to blocks B1, B2 and B1, respectively. The external event EEx enters the block B1 and the corresponding processor locks the block B1 before starting execution in the block, as indicated by the diagonal line across the block B1. Next, the external event EEy enters the block B2 and the corresponding processor locks the block B2. As indicated by the time axis (t) of Fig. 6, the external event EEz directed to block B1 comes after the external event EEx which has already entered block B1 and locked that block. Accordingly, the processing of external event EEz has to wait until block B1 is released.

Locking, however, might give rise to deadlock conditions in which two processors indefinitely wait for each other to release variables mutually required by the processors in execution of their current tasks. It is therefore desirable either to avoid deadlocks, or to detect them and perform roll-back with guarantee of progress.

It is possible to avoid deadlocks by seizing all the blocks required in the execution of a complete task, also referred to as a job, at the beginning of the job, as opposed to seizing/locking the blocks as required during the execution. However, it may not always be possible to know all the required blocks for a given job in advance, although non-run time inputs using compiler analysis might provide information to minimize deadlocks, for example by seizing at least those blocks that consume a higher fraction of the processing time within the job. An efficient way of minimizing deadlocks is to seize such a high usage block before starting execution irrespective of whether it is the next block required in the processing or not. It is always a good idea to seize blocks that will almost surely be required by a job, especially those with high usage, and seize the rest of the blocks as and when required.

Seizing the blocks as required during execution is prone to deadlocks as explained above, thus making it necessary to detect and resolve the deadlocks. It is advantageous to detect deadlocks as early as possible, and according to the invention deadlock detection could be almost immediate. Since all "overhead processing" takes place between two jobs, deadlock detection will be evident while acquiring "resources" for a later job that will cause a deadlock. This is accomplished by checking if one of the resources required by the job under consideration is held by some processor, and then verifying whether that processor is waiting on a resource held by the processor with the job under consideration – for example by using flags per blocks.

Minimizing deadlocks will normally also have an impact on the scheme for roll-back and progress. The lower the deadlock frequency, the simpler the roll-back scheme, as one does not have to bother about the efficiency of rare roll-backs. On the other hand, if the deadlock frequency is relatively high it is important to have an efficient roll-back scheme.

The basic principle for roll-back is to release all the held resources, go back to the beginning of one of the jobs involved in causing the deadlock, undoing all changes made in the execution up to that point, and restart the rolled-back job later in such a way, or after such a delay, that progress can be guaranteed without compromising the efficiency. This generally means that the roll-back scheme is not allowed to cause recurring deadlocks resulting in roll-backs of the same job by restarting it immediately, nor should the delay before starting the rolled-back job be too long. However, when the execution times of the jobs are very short, simply selecting the "later" job causing the deadlock for roll-back should be adequate.

Collision detection as a means of assuring data consistency

When implementing collision detection for the purpose of assuring data consistency, the software tasks are executed in parallel by the multiple processors, and access collisions are detected so that one or more executed tasks for which collisions are detected can be rolled-back and restarted.

Preferably, each processor marks the use of variables in the shared memory while executing a task, thus enabling variable access collisions to be detected.

At its very basic level, the marker method consists of marking the use of individual variables in the shared memory. However, by marking larger areas instead of individual data, a more coarse-grained collision check is realized. One way of implementing a more coarse-grained collision check is to utilize standard memory management techniques including paging. Another way is to mark groupings of variables, and it has turned out be particularly efficient to mark entire records including all record variables in the records, instead of marking individual record variables. It is however important to choose "data areas" in such a way that if a job uses a given data area then the probability of some other job using the same area should be very low. Otherwise, the coarse-grained data-area marking may in fact result in a higher roll-back frequency.

Fig. 7 illustrates the use of variable marking to detect access collisions in an object-oriented software design. The shared memory 12 is organized into blocks B1 to Bn as described above in connection with Fig. 4, and a number of processors P1 to P3 are connected to the shared memory 12. Fig. 7 shows two blocks, block B2 and block B4, in more detail. In this particular realization of the marker method, each global variable GV1 to GVn and each record R1 to Rn in a block is associated with a marker field as illustrated in Fig. 7.

The marker field has 1 bit per processor connected to the shared memory system, and hence in this case, each marker field has 3 bits. All bits are reset at start, and each processor sets its own bit before accessing (read or write) a variable or record, and then reads the entire marker field for evaluation. If there is any other bit that is set in the marker field, then a collision is imminent, and the processor rolls back the task being executed, undoing all changes made up to that point in the execution including resetting all the corresponding marker bits. On the other hand, if no other bit is set then the processor continues execution of the task. Each processor records the address of each variable accessed during execution, and uses the recorded address(es) to reset its own bit in each of the corresponding marker fields at the end of execution of a task.

In order to be able to do a roll-back when a collision is detected, it is necessary to keep a copy of all modified variables (the variable states before modification) and their addresses during execution of each job. This allows restoration of the original state(s) in case of a roll-back.

In Fig. 7, the processor P2 needs to access the global variable GV1, and sets its own bit at the second position of the marker field associated with GV1, and then reads the entire marker field. In this case, the field (110) contains a bit set by processor P1 and a bit set by processor P2, and consequently an imminent variable access collision is detected. The processor P2 rolls back the task being executed. Correspondingly, if processor P2 needs to access the record R2, it sets its own bit at the second position, and then reads the entire marker field. The field (011) contains a bit set by P2 and a bit set by P3, and consequently a record access collision is detected, and the processor P2 rolls back the task being executed. When processor P3 needs to access the record R1, it first sets its own bit in the third position of the associated marker field, and then reads the entire field for evaluation. In this case, no other bits are set so the processor P3 is allowed to access the record for a read or write. Preferably, each marker field will have two bits per processor, one bit for write and one bit for read so as to reduce unnecessary roll-backs, for example on variables that are mostly read.

Another approach for collision detection is referred to as the address comparison method, where read and write addresses are compared at the end of a task. The main difference compared to the marker method is that accesses by other processors are generally not checked during execution of a task, only at the end of a task. An example of a specific type of checking unit implementing an address comparison method is disclosed in our international patent application WO 88/02513.

Reuse of existing application software

Existing sequentially programmed application software normally represents large investments, and for single-processor systems, such as single-processor nodes at the highest level of hierarchical processing systems, thousands or millions of lines of software code already exist. By automatically transforming the application software via recompilation or equivalent, and assuring data consistency when the application software is executed on multiple processors, all of the software code can be migrated to and reused in a multiprocessor environment, thus saving time and money.

Fig. 8A illustrates a prior art single-processor system from a stratified viewpoint. At the bottom layer, the processor P1 such as a standard microprocessor can be found. The next level includes the operating system, and then comes the virtual machine, which interprets the application software found at the top level.

Fig. 8B illustrates a multiprocessor system from a stratified viewpoint. At the bottom level, multiple shared-memory processors P1 and P2 implemented as standard off-the–shelf microprocessors are found. Then comes the operating system. The virtual machine, which by way of example may be an APZ emulator running on a SUN work station, a compiling high-performance emulator such as SIMAX or the well-known Java Virtual Machine, is modified for multiprocessor support and data-consistency related support. The sequentially programmed application software is generally transformed by simply adding code for data-consistency related support by post-processing the object code or recompiling blocks/classes if compiled, or modifying the interpreter if interpreted.

In the case of collision detection based on variable markings, the following steps may be taken to enable migration of application software written for a single-processor system to a multiprocessor environment. Before each write access to a variable, code for storing the address and original state of the variable is inserted into the application software to enable proper roll-back. Before each read and write access to a variable, code for setting marker bits in the marker field, checking the marker field as well as for storing the address of the variable is inserted into the software. The application software is then recompiled or reinterpreted, or the object code is post-processed. The hardware/operating system/virtual machine is also modified to give collision detection related support, implementing roll-back and resetting of marker fields. Accordingly, if a collision is detected when executing code for checking the marker field, the control is normally transferred to the hardware/operating system/virtual machine, which performs roll-back using the stored copy of the modified variables. In addition, at the end of a job, the hardware/operating system/virtual machine normally takes over and resets the relevant bit in each of the marker fields given by the stored addresses of variables that have been accessed by the job.

Note that static analysis of code might allow minimizing the insertion of new code. For example, instead of before each read and write as described above, the code insertions could be done in fewer places in such a way that the final objective is met.

It should though be understood that if the multiple processors are implemented as specialized hardware of proprietary design, the application software can be migrated directly to the multiprocessor environment.

Fig. 9 is a schematic diagram of a communication system in which one or more processing systems according to the invention are implemented. The communication system 100 may support different bearer service networks such as PSTN (Public Switched Telephone Network), PLMN (Public Land Mobile Network), ISDN (Integrated Services Digital Network) and ATM (Asynchronous Transfer Mode) networks. The communication system 100 basically comprises a number of switching/routing nodes 50-1 to 50-6 interconnected by physical links that are normally grouped into trunk groups. The switching nodes 50-1 to 50-4 have access points to which access terminals, such as telephones 51-1 to 51-4 and computers 52-1 to 52-4, are connected via local exchanges (not shown). The switching node 50-5 is connected to a Mobile Switching Center (MSC) 53. The MSC 53 is connected to two Base Station Controllers (BSCs) 54-1 and 54-2, and a Home Location Register (HLR) node 55. The first BSC 54-1 is connected to a number of base stations 56-1 and 56-2 communicating with one or more mobile units 57-1 and 57-2. Similarly, the second BSC 54-2 is connected to a number of base stations 56-3 and 56-4 communicating with one or more mobile units 57-3. The switching node 50-6 is connected to a host computer 58 provided with a data base system (DBS). User terminals connected to the system 100, such as the computers 52-1 to 52-4, can request data base services from the data base system in the host computer 58. A server 59, especially a Java server, is connected to the switching/routing node 50-4. Private networks such as business networks (not shown) may also be connected to the communication system of Fig. 1.

The communication system 100 provides various services to the users connected to the network. Examples of such services are ordinary telephone calls in PSTN and PLMN, message services, LAN interconnects, Intelligent Network (IN) services, ISDN services, CTI (Computer Telephony Integration) services, video conferences, file transfers, access to the so-called Internet, paging services, video-on-demand and so on.

According to the invention, each switching node 50 in the system 100 is preferably provided with a processing system 1-1 to 1-6 according to the first or second aspect of the invention (possibly a combination of the two aspects in the form of a matrix processing system), which handles events such as service requests and inter-node communication. A call set-up for example requires the processing system to execute a sequence of jobs. This sequence of jobs defines the call set-up service on the processor level. A processing system according to the invention is preferably also arranged in each one of the MSC 53, the BSCs

54-1 and 54-2, the HLR node 55 and the host computer 58 and the server 59 of the communication system 100.

Although the preferred use of the invention is in high-level processor nodes of hierarchical processing systems, those of ordinary skill in the art will appreciate that the above described aspects of the invention are applicable to any event-driven processing where event-flow concurrency can be identified.

The term event-based system includes but is not limited to telecommunication, data communication and transaction-oriented systems.

The term shared-memory processors is not limited to standard off-the-shelf microprocessors, but includes any type of processing units, such as SMPs and specialized hardware, operating towards a common memory with application software and data accessible to all processing units. This also includes systems where the shared memory is distributed over several memory units and even systems with asymmetrical access where the access times to different parts of the distributed shared memory for different processors could be different.

The embodiments described above are merely given as examples, and it should be understood that the present invention is not limited thereto. Further modifications, changes and improvements which retain the basic underlying principles disclosed and claimed herein are within the scope and spirit of the invention.

CLAIMS

1. An event-based hierarchical distributed processing system (1) having a plurality of processor nodes distributed over a number of levels of the system hierarchy,

characterized in that at least one high-level processor node (10) within the hierarchical processing system (1) comprises:

multiple shared-memory processors (11);

means (14) for mapping external events arriving to the processor node onto the processors such that the external event flow is divided into a number of non-commuting categories of events and each non-commuting category of events is assigned to a predetermined set of the shared-memory processors for processing by the processor(s) of that set; and

means (15) for assuring data consistency when global data of the shared memory (12) are manipulated by the processors.

2. The hierarchical distributed processing system according to claim 1, characterized in that each processor set is in the form of a single processor.

3. The hierarchical distributed processing system according to claim 1, characterized in that at least one processor set is in the form of an array of processors operating as a multiprocessor pipeline having a number of processor stages, where each event of the non-commuting category assigned to the processor set is processed in slices as a chain of events which are executed in different processor stages of the pipeline.

4. The hierarchical distributed processing system according to claim 3, characterized in that events requiring input data from a predetermined data area in the shared memory (12) are mapped by the mapping means (14, 18) to one and the same predetermined processor set.

5. The hierarchical distributed processing system according to claim 1, characterized in that the non-commuting categories are groupings of events where the order of events must be preserved within a category, but where there is no ordering requirement on processing events of different categories.

6. The hierarchical distributed processing system according to claim 1, characterized in that the high-level processor node further comprises means for feeding events generated by a processor set to the same processor set.

7. The hierarchical distributed processing system according to claim 1, characterized in that a non-commuting category is defined by events from a predetermined source (S1/S2).

8. The hierarchical distributed processing system according to claim 7, characterized in that the source (S1/S2) is an input port, lower-level processor node or a hardware device connected to the hierarchical distributed processing system.

9. The hierarchical distributed processing system according to claim 1, characterized in that the data consistency means (15) comprises means for locking a global variable, in the shared memory, to be used by a software task executed in response to an event, and means for releasing the locked global variable at the end of execution of the task.

10. The hierarchical distributed processing system according to claim 9, characterized in that the data consistency means (15) further comprises means for releasing a locked global variable of one of two mutually locking tasks and restarting that task after an appropriate delay.

11. The hierarchical distributed processing system according to claim 1, characterized in that software in the shared memory (12) includes a number of software blocks (B1 to Bn), and each one of the processors executes a software

task including a software block in response to an event and each processor comprises means, forming part of the data consistency assuring means (15), for locking at least the global data of the software block before starting execution of the task such that only the processor that has locked the block can access global data within that block.

12. The hierarchical distributed processing system according to claim 11, characterized in that the locking means locks the entire software block before starting execution of the corresponding task and releases the locked block at the end of execution of the task.

13. The hierarchical distributed processing system according to claim 11, characterized in that the locking means seizes at least those blocks required by a software task that consume a high fraction of the processing time within the task before starting execution of the task to minimize deadlock conditions.

14. The hierarchical distributed processing system according to claim 11, characterized in that the high-level processor node comprises means for detecting a deadlock condition, and means for releasing a block locked by one of the waiting processors and restarting the software task executed by that processor after an appropriate delay so as to ensure progress.

15. The hierarchical distributed processing system according to claim 14, characterized in that the deadlock detecting means comprises means for checking whether a variable required by a software task under consideration is locked by another processor, and means for verifying whether that other processor is waiting on a variable locked by the processor with the task under consideration.

16. The hierarchical distributed processing system according to claim 1, characterized in that the multiple processors (11) independently process events to execute a number of corresponding software tasks in parallel, and

the data consistency assuring means (15) comprises means for detecting collisions between parallel tasks, and means for undoing and restarting a task for which a collision is detected.

17. The hierarchical distributed processing system according to claim 16, characterized in that each processor comprises means for marking the use of variables in the shared memory and the collision detecting means includes means for detecting variable access collisions based on the markings.

18. The hierarchical distributed processing system according to claim 16, characterized in that software in the shared memory (12) includes a number of software blocks (B1 to Bn), and each one of the multiple processors executes a software task including a software block in response to an event and each processor comprises means for marking the use of variables within the block and the collision detecting means includes means for detecting variable access collisions based on the markings.

19. The hierarchical distributed processing system according to claim 1, characterized in that the high-level processor node (10) further comprises parallel event queues (16), a queue towards each processor set, and the mapping means (14) maps the external events onto the event queues based on information included in each of the external events.

20. An event-based hierarchical distributed processing system (1) having a plurality of processor nodes distributed over a number of levels of the system hierarchy,

characterized in that at least one high-level processor node (10) within the hierarchical processing system comprises:

multiple shared-memory processors (11) operating as a multiprocessor pipeline having a number of processor stages, each event arriving to the multiprocessor pipeline being processed in slices as a chain of events which are executed in different processor stages of the pipeline;

means (15) for assuring data consistency when global data of the shared memory (12) are manipulated by the processors.

21. The hierarchical distributed processing system according to claim 20, characterized in that software in the shared memory (12) includes a number of software blocks (B1 to Bn), and each event is directed to one of the software blocks; and

the multiprocessor pipeline is realized by means (17, 18) for allocating a cluster (CL) of blocks to each one of the processors in a load-balanced way, and means (17, 18) for mapping events onto the processors according to the allocation made by the allocating means.

22. The hierarchical distributed processing system according to claim 20, characterized in that the data consistency means (15) comprises means for locking a global variable, in the shared memory, to be used by a task executed by a processor in response to an event, and means for releasing the locked global variable at the end of execution of the task.

23. The hierarchical distributed processing system according to claim 22, characterized in that the data consistency means (15) further comprises means for releasing a global variable of one of two mutually locking tasks and restarting that task after an appropriate delay.

24. The hierarchical distributed processing system according to claim 20, characterized in that software in the shared memory (12) includes a number of software blocks (B1 to Bn), and each one of the processors executes a software task including a software block in response to an event and each processor comprises means, forming part of the data consistency assuring means (15), for locking at least the global data of the software block before starting execution of the task such that only the processor that has locked the block can access global data within that block.

25. The hierarchical distributed processing system according to claim 24, characterized in that the locking means locks the entire software block before starting execution of the corresponding task and releases the locked block at the end of execution of the task.

26. The hierarchical distributed processing system according to claim 24, characterized in that the locking means seizes at least those blocks required by a software task that consume a high fraction of the processing time of the task before starting execution of the task to minimize deadlock conditions.

27. The hierarchical distributed processing system according to claim 24, characterized in that the high-level processor node (10) comprises means for detecting a deadlock condition, and means for releasing a block locked by one of the waiting processors and restarting the software task executed by that processor after an appropriate delay so as to ensure progress.

28. The hierarchical distributed processing system according to claim 27, characterized in that the deadlock detecting means comprises means for checking whether a variable required by a software task under consideration is locked by another processor, and means for verifying whether that other processor is waiting on a variable locked by the processor with the task under consideration.

29. The hierarchical distributed processing system according to claim 20, characterized in that the multiple processors (11) independently process events to execute a number of corresponding software tasks in parallel, and the data consistency assuring means (15) comprises means for detecting collisions between parallel tasks and means for undoing and restarting a task for which a collision is detected.

30. The hierarchical distributed processing system according to claim 29, characterized in that each processor comprises means for marking the use of variables in the shared memory (12) and the collision detecting means includes means for detecting variable access collisions based on the markings.

31. A processing method in an event-based hierarchical distributed processing system (1) having a plurality of processor nodes distributed over a number of levels of the system hierarchy,

characterized by:

providing multiple shared-memory processors (11) in at least one high-level processor node (10) within the hierarchical processing system (1);

dividing an external flow of events to the processor node into a number of non-commuting categories (NCCs) of events based on an event-flow concurrency identified in the system;

mapping the NCCs towards the processors such that each NCC of events is assigned to a predetermined set of the multiple processors for processing by the processor(s) of that set; and

assuring data consistency when global data of the shared memory (12) are manipulated by the processors so that only one of the processors accesses given global data at a time.

32. The processing method according to claim 31, characterized in that the NCCs are groupings of events where the order of events must be preserved within a category, but where there is no ordering requirement on processing events of different categories.

33. The processing method according to claim 31, characterized by operating at least one processor set as a multiprocessor pipeline having a number of processor stages, where each event of the non-commuting category assigned to the processor set is processed in slices as a chain of events which are executed in different processor stages of the pipeline.

34. The processing method according to claim 31, characterized by feeding events generated by a processor set to the same processor set.

35. The processing method according to claim 31, characterized in that the step of assuring data consistency includes locking a global variable, in the shared memory, to be used by a software task executed in response to an event, and releasing the locked global variable at the end of execution of the software task.

36. The processing method according to claim 35, characterized in that the step of assuring data consistency further includes releasing a global variable of one of two mutually locking tasks and restarting that task after an appropriate delay.

37. The processing method according to claim 31, characterized in that software in the shared memory (12) includes a number of software blocks and each one of the processors executes a software task including a software block in response to an event, and the step of assuring data consistency includes locking at least the global data of a software block before execution by one of the processors such that only that processor can access global data within the block.

38. The processing method according to claim 37, characterized in that the entire software block is locked before starting execution of the corresponding task, and the locked block is released at the end of execution of the task.

39. The processing method according to claim 37, characterized by seizing all the blocks required in a software task before starting execution of the task to avoid so-called deadlock conditions.

40. The processing method according to claim 37, characterized by detecting a deadlock condition, and releasing a block locked by one of the waiting processors and restarting the software task executed by that processor after a predetermined delay so as to ensure progress.

41. The processing method according to claim 31, characterized in that the processors, in response to events, execute a number of corresponding software tasks in parallel, and the step of assuring data consistency includes detecting access collisions, and undoing and restarting a task for which a collision is detected.

42. The processing method according to claim 41, characterized in that each processor marks the use of variables in the shared memory and the collision detecting step includes detecting variable access collisions based on the markings.

43. The processing method according to claim 31, characterized in that the method further comprises the step of migrating application software for a single-processor system to the multiple shared-memory processors for execution thereby.

44. A processing method in an event-based hierarchical distributed processing system (1) having a plurality of processor nodes distributed over a number of levels of the system hierarchy,

characterized by:

providing multiple shared-memory processors (11) in at least one high-level processor node (10) within the hierarchical processing system (1);

operating the multiple processors (11) as a multiprocessor pipeline having a number of processor stages, at least one of the events arriving to the multiprocessor pipeline being processed in slices as a chain of events which are executed in different processor stages of the pipeline; and assuring data consistency when global data of the shared memory are manipulated by the multiprocessors so that only one of the processors accesses such global data at a time.

45. The processing method according to claim 44, characterized in that software in the shared memory (12) includes a number of software blocks (B1 to Bn), and each event is directed to one of the software blocks, and the step of operating the processors as a multiprocessor pipeline includes allocating a cluster (CL) of blocks to each one of the processors in a load balanced way, and mapping events onto the processors according to the allocation.

46. The processing method according to claim 44, characterized in that the step of assuring data consistency includes locking a global variable, in the shared memory, to be used by a software task executed in response to an event, and releasing the locked global variable at the end of execution of the software task.

47. The processing method according to claim 46, characterized in that the step of assuring data consistency further includes releasing a global variable of one of two mutually locking tasks and restarting that task after an appropriate delay.

48. The processing method according to claim 44, characterized in that software in the shared memory (12) includes a number of software blocks and each one of the processors executes a software task including a software block in response to an event, and the step of assuring data consistency includes locking at least the global data of a software block before execution by one of the processors such that only that processor can access global data within the block.

49. The processing method according to claim 48, characterized in that the entire software block is locked before starting execution of the corresponding task, and the locked block is released at the end of execution of the task.

50. The processing method according to claim 48, characterized by seizing all the blocks required in a software task before starting execution of the task to avoid so-called deadlock conditions.

51. The processing method according to claim 48, characterized by detecting a deadlock condition, and releasing a block locked by one of the waiting processors and restarting the software task executed by that processor after a predetermined delay so as to ensure progress.

52. The processing method according to claim 44, characterized in that the processors execute, in response to events, a number of corresponding software tasks in parallel, and the step of assuring data consistency includes detecting access collisions between parallel tasks, and undoing and restarting a task for which a collision is detected.

53. The processing method according to claim 52, characterized in that each processor marks the use of variables in the shared memory and the collision detecting step includes detecting variable access collisions based on the markings.

54. The processing method according to claim 44, characterized in that the method further comprises the step of migrating application software for a single-processor system to the multiple shared-memory processors for execution thereby.

55. An event-driven processing system comprising:

multiple shared-memory processors (11) for executing a number of jobs in parallel;

a mapper (14) for mapping concurrent categories of external, self-contained event signals onto the multiple processors (11) for concurrent execution of corresponding jobs;

a collision detector for detecting access collisions between parallel jobs when global data of the shared memory are manipulated by the processors; and

means for undoing and restarting a job for which a collision is detected so that data consistency is assured.

56. An event-driven processing system according to claim 55, further comprising means for feeding jobs generated by a processor to the same processor.

57. An event-driven processing system comprising:

multiple shared-memory processors (11) for executing a number of jobs in parallel, the processors operating as a multiprocessor pipeline having a number of processor stages, where each external event signal arriving to the multiprocessor pipeline is processed in slices as a chain of jobs which are executed in different processor stages of the pipeline;

a collision detector for detecting access collisions between parallel jobs when global data of the shared memory are manipulated by the processors; and

means for undoing and restarting a job for which a collision is detected so that data consistency is assured.

58. A processing system comprising:

a shared memory (12) including software in the form of a number of software blocks;

multiple shared-memory processors (11) for parallel execution of jobs, each job being associated with at least one of the software blocks;

means (17, 18) for allocating a cluster (CL) of the software blocks to each of the processors;

means (17, 18) for distributing the jobs to the processors for execution based on the allocation made by the allocating means; and

means (15) for assuring data consistency when global data of the shared memory are manipulated by the multiprocessors.

59. A communication system (100) comprising an event-based hierarchical distributed processing system (1) having a plurality of processor nodes distributed over a number of levels of the system hierarchy,

characterized in that at least one high-level processor node (10) within the hierarchical processing system comprises:

multiple shared-memory processors (11);

means (14) for mapping external events arriving to the processor node onto the processors such that the external event flow is divided into a number of non-commuting categories of events and each non-commuting category of events is assigned to a predetermined set of the shared-memory processors for processing by the processor(s) of that set; and

means (15) for assuring data consistency when global data of the shared memory (12) are manipulated by the processors.

60. A communication system (100) comprising an event-based hierarchical distributed processing system (1) having a plurality of processor nodes distributed over a number of levels of the system hierarchy,

characterized in that at least one high-level processor node (10) within the hierarchical processing system comprises:

multiple shared-memory processors (12) operating as a multiprocessor pipeline having a number of processor stages, each event arriving to the multiprocessor pipeline being processed in slices as a chain of events which are executed in different processor stages of the pipeline; and

means (15) for assuring data consistency when global data of the shared memory (12) are manipulated by the processors.

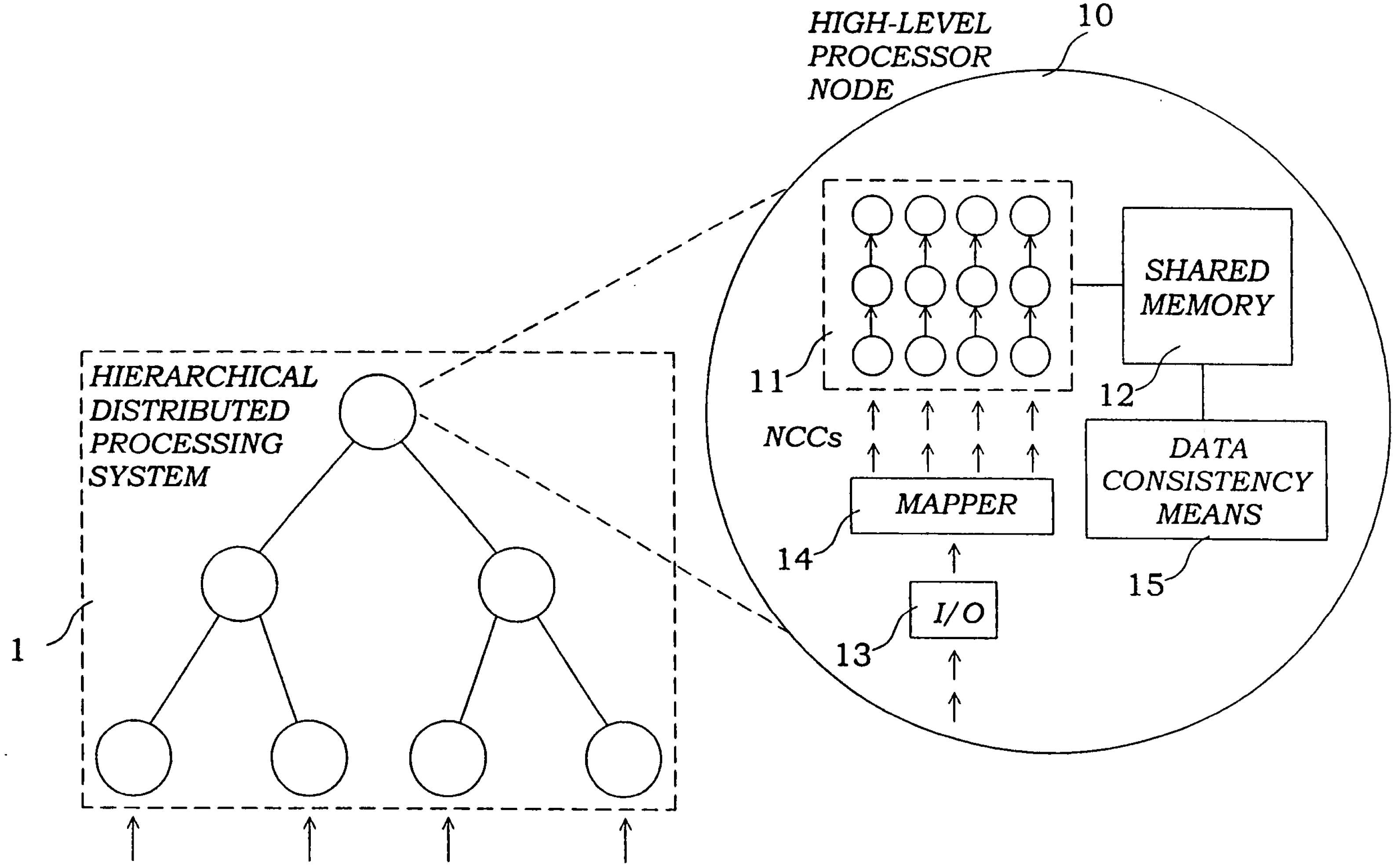

HIGH-LEVEL
PROCESSOR
NODE
10
11
NCCs
SHARED
MEMORY
12
MAPPER
14
DATA
CONSISTENCY
MEANS
15
I/O
13
HIERARCHICAL
DISTRIBUTED
PROCESSING
SYSTEM
1

Fig. 1

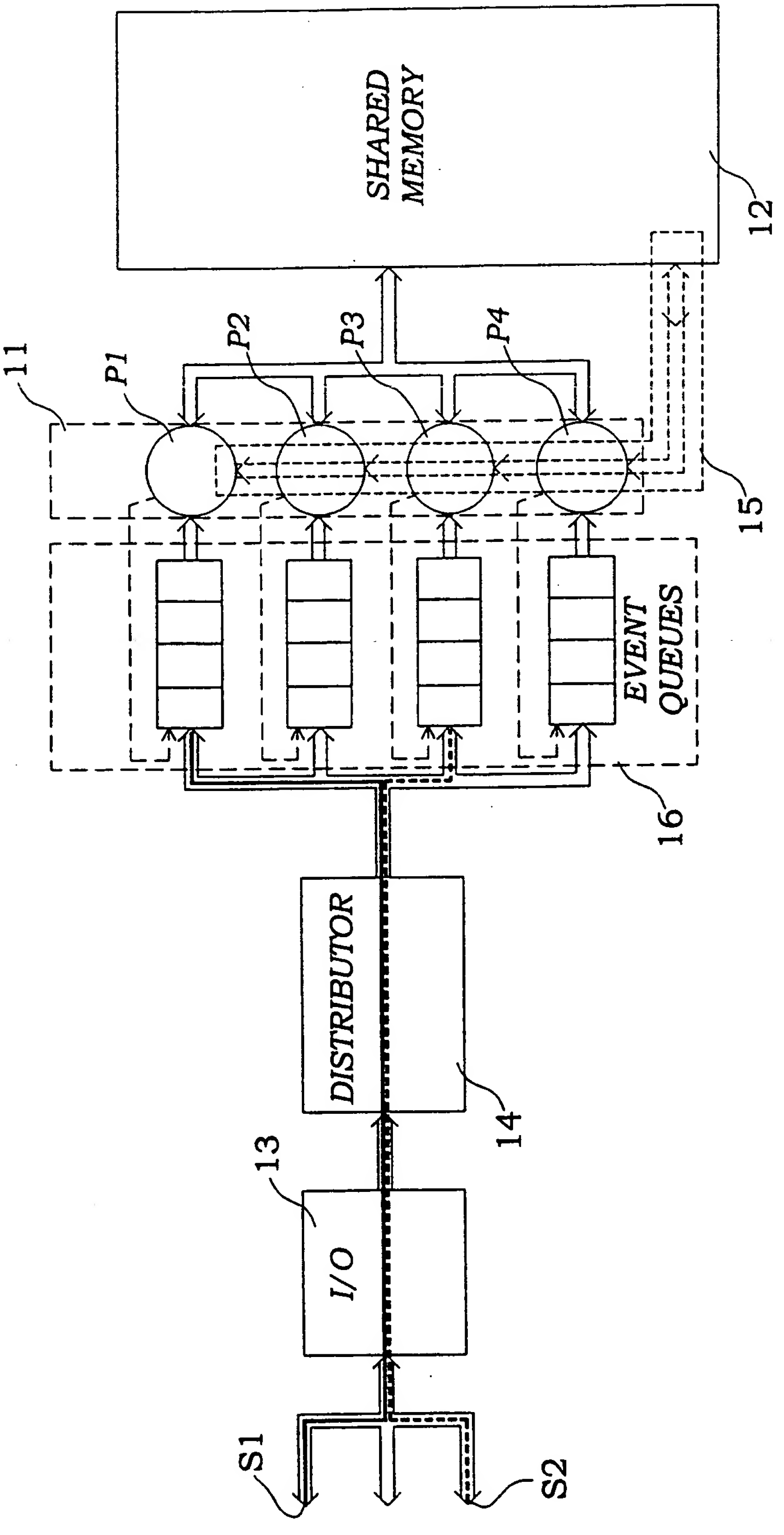

SHARED MEMORY
12
11
P1
P2
P3
P4
15
EVENT QUEUES
16
DISTRIBUTOR
14
I/O
13
S1
S2

Fig. 2

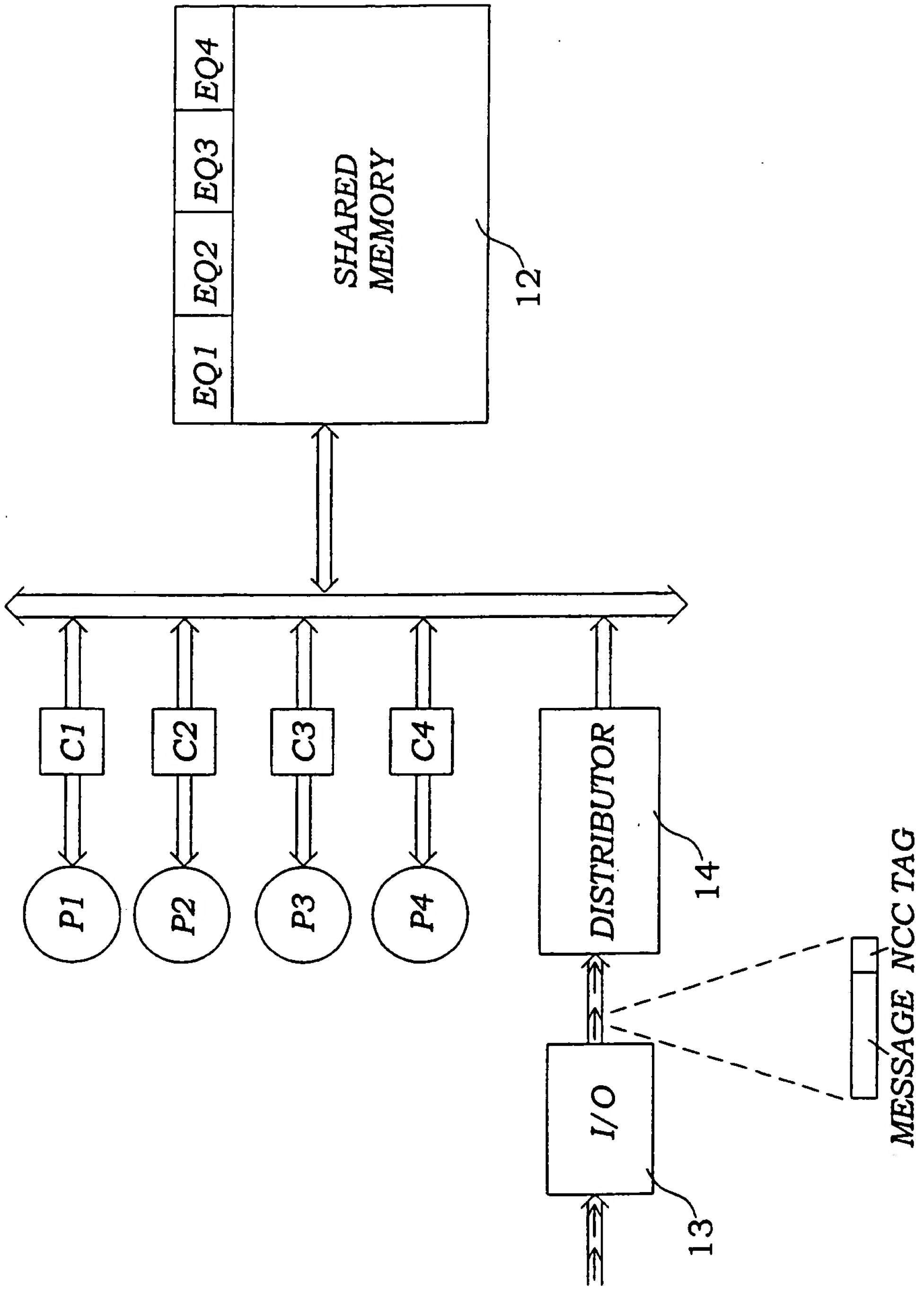

EQ1
EQ2
EQ3
EQ4
SHARED MEMORY
12
C1
C2
C3
C4
P1
P2
P3
P4
DISTRIBUTOR
14
I/O
13
MESSAGE
NCC TAG

Fig. 3

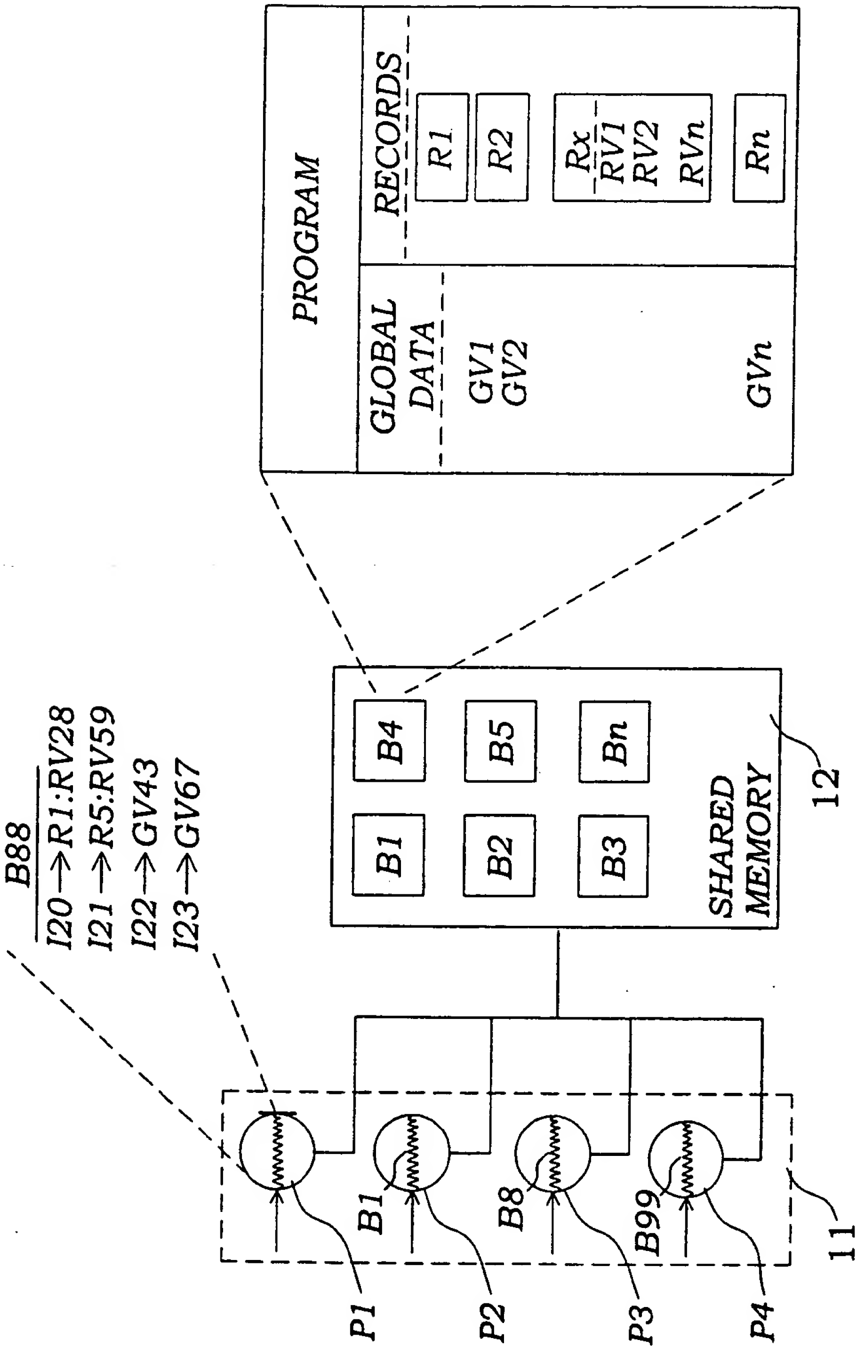

B88
I20→R1:RV28
I21→R5:RV59
I22→GV43
I23→GV67
B1
B8
B99
P1
P2
P3
P4
11
B1
B2
B3
B4
B5
Bn
SHARED MEMORY
12
PROGRAM
GLOBAL DATA
GV1
GV2
GVn
RECORDS
R1
R2
Rx
RV1
RV2
RVn
Rn

Fig. 4

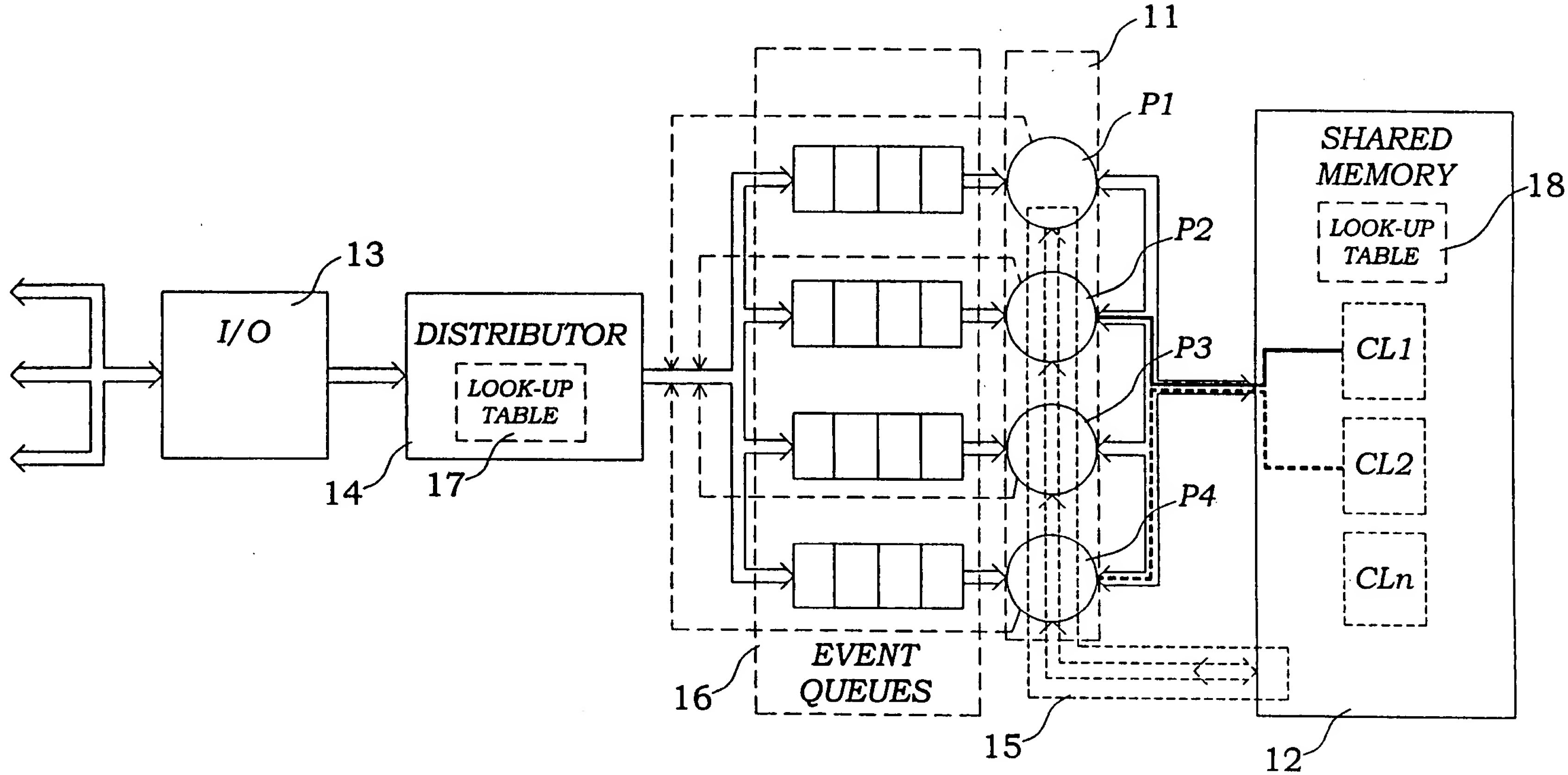

13
I/O
DISTRIBUTOR
LOOK-UP TABLE
14
17
EVENT QUEUES
16
11
P1
P2
P3
P4
15
SHARED MEMORY
LOOK-UP TABLE
18
CL1
CL2
CLn
12

Fig. 5A

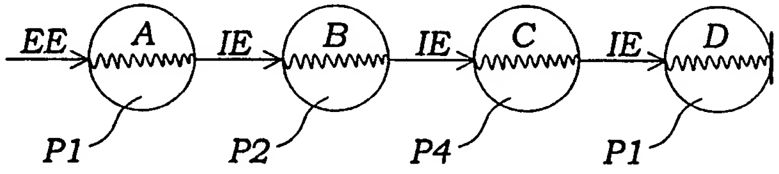

EE
A
IE
B
IE
C
IE
D
P1
P2
P4
P1

Fig. 5B

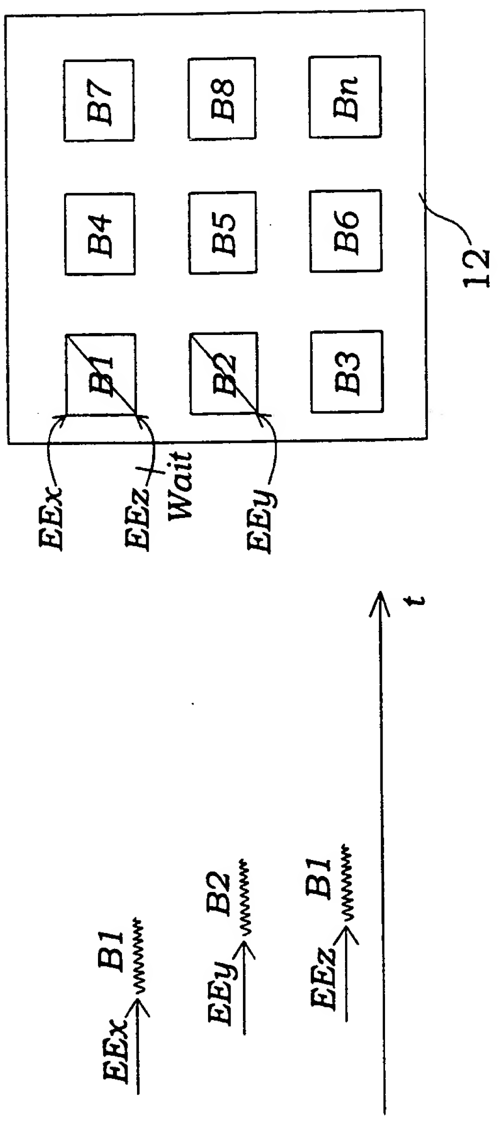


Fig. 6

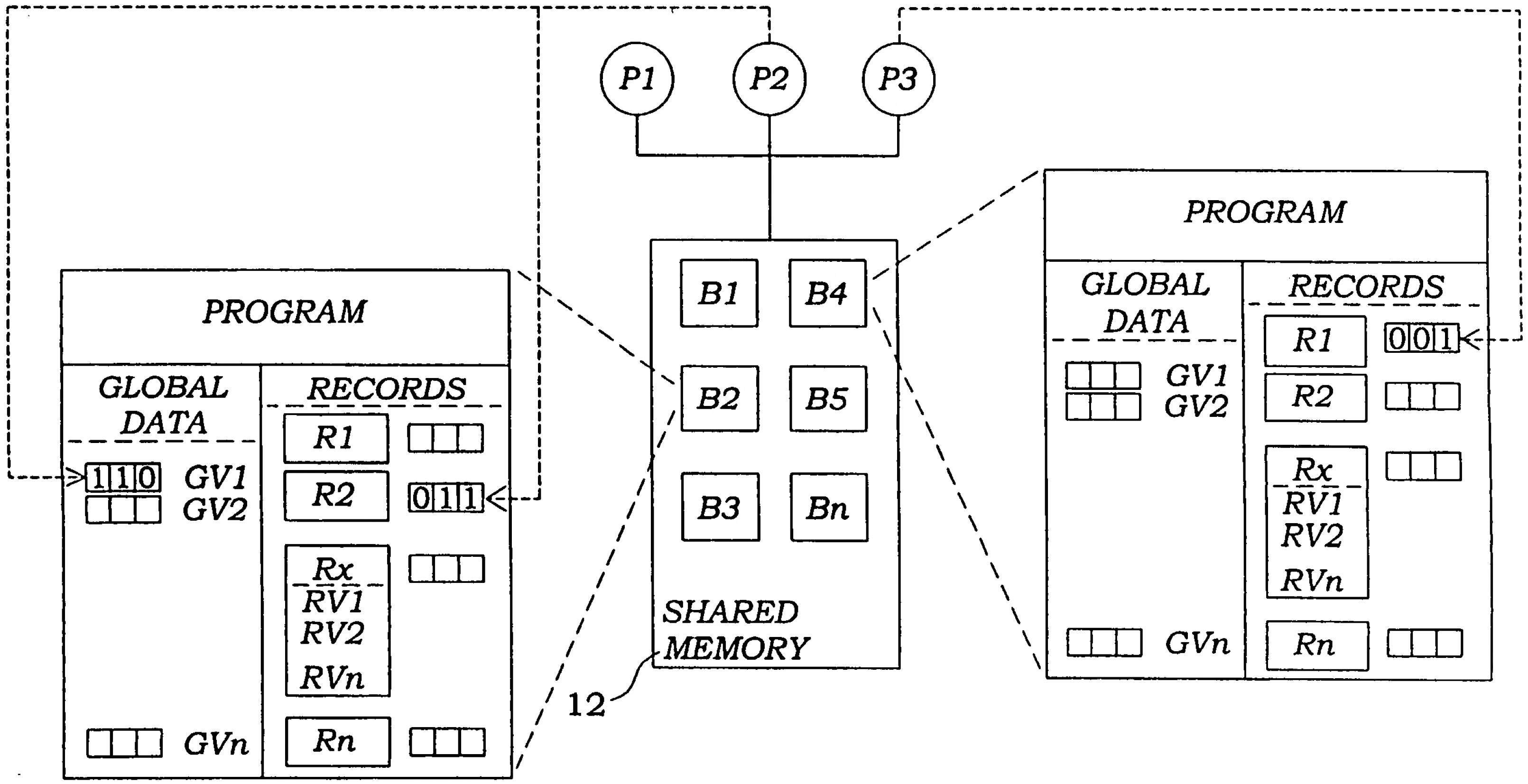

P1
P2
P3
PROGRAM
GLOBAL DATA
RECORDS
R1
R2
Rx
RV1
RV2
RVn
Rn
1 1 0
0 1 1
GV1
GV2
GVn
B1
B4
B2
B5
B3
Bn
SHARED MEMORY
12
PROGRAM
GLOBAL DATA
RECORDS
R1
0 0 1
R2
Rx
RV1
RV2
RVn
Rn
GV1
GV2
GVn

Fig. 7

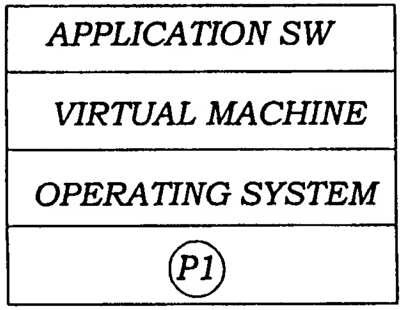


Fig. 8A
(Prior art)

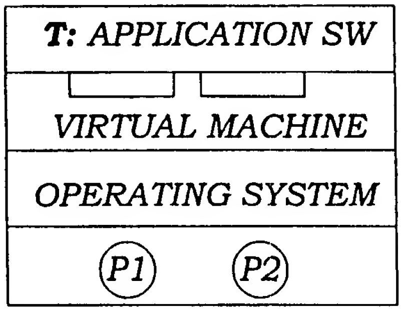


Fig. 8B

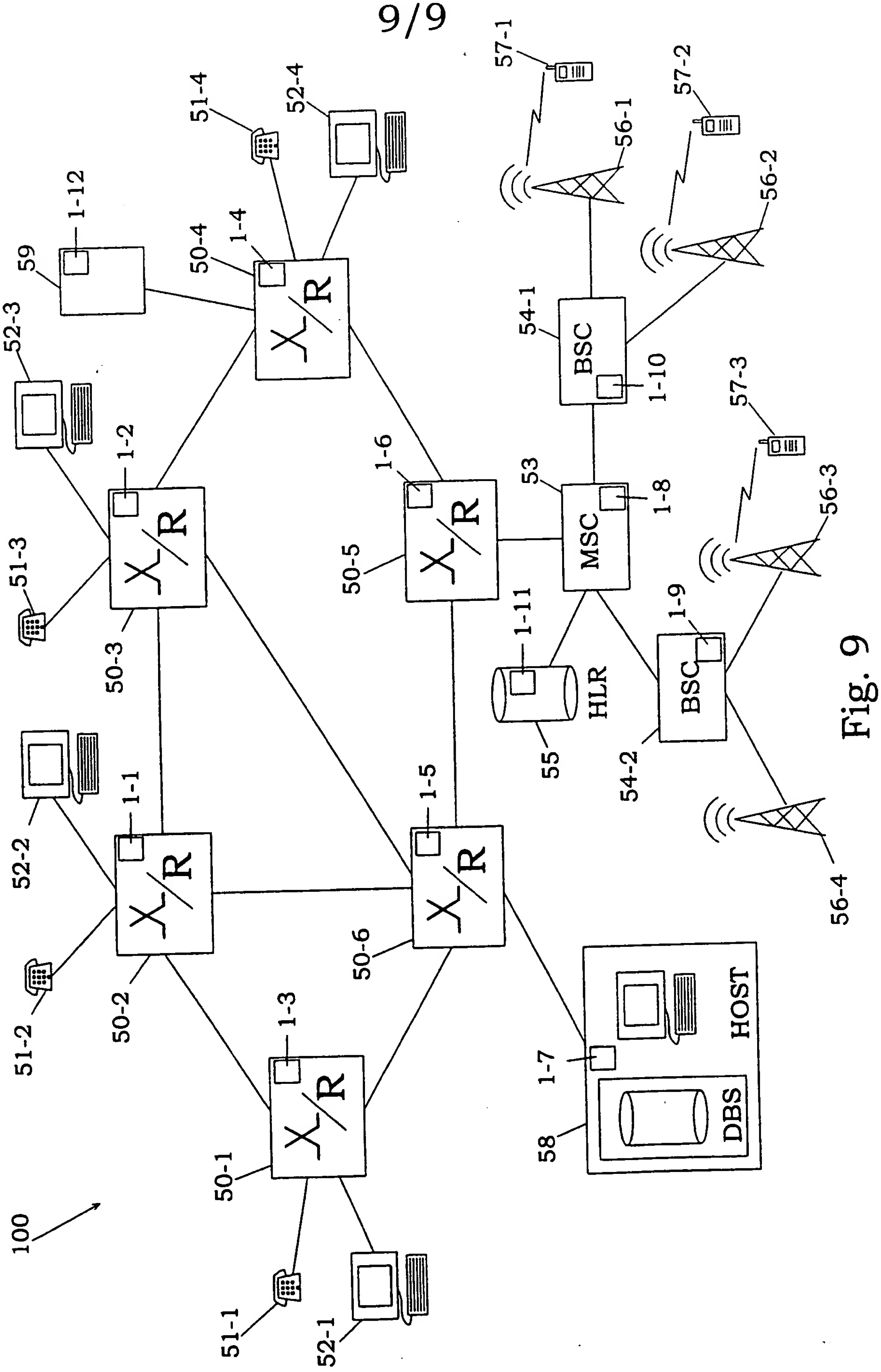

100
50-1
1-3
51-1
52-1
50-2
1-1
51-2
52-2
50-3
1-2
51-3
52-3
50-4
1-4
51-4
52-4
59
1-12
50-5
1-6
50-6
1-5
53
MSC
1-8
54-1
BSC
1-10
54-2
BSC
1-9
55
HLR
1-11
56-1
56-2
56-3
56-4
57-1
57-2
57-3
58
HOST
DBS
1-7
X/R

Fig. 9

INTERNATIONAL SEARCH REPORT

Information on patent family members

02/12/99

International application No.
PCT/SE 99/02064

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5379428 A	03/01/95	NONE	
US 5511172 A	23/04/96	JP 5224927 A	03/09/93
US 5072364 A	10/12/91	AU 631874 B	10/12/92
		AU 5504790 A	29/11/90
		CA 2016254 A	23/11/90
		EP 0399760 A	28/11/90
		EP 0939364 A	01/09/99
		JP 2846406 B	13/01/99
		JP 3116235 A	17/05/91
US 5287467 A	15/02/94	DE 69229198 D	00/00/00
		EP 0509245 A,B	21/10/92
		JP 2051694 C	10/05/96
		JP 5143336 A	11/06/93
		JP 7085223 B	13/09/95
		US 5377336 A	27/12/94
US 5848257 A	08/12/98	NONE	
US 5812839 A	22/09/98	AT 184407 T	15/09/99
		DE 69420540 D	00/00/00
		EP 0661625 A,B	05/07/95
		SE 0661625 T3	
		SG 52391 A	28/09/98
WO 9931589 A1	24/06/99	AU 1911099 A	05/07/99